FEEBO: An Empirical Evaluation Framework for Malware Behavior Obfuscation

Sebastian Banescu, Tobias Wüchner, Marius Guggenmos, Martín Ochoa, and Alexander Pretschner

Technische Universität München, Germany

Abstract. Program obfuscation is increasingly popular among malware creators. Objectively comparing different malware detection approaches with respect to their resilience against obfuscation is challenging. To the best of our knowledge, there is no common empirical framework for evaluating the resilience of malware detection approaches w.r.t. behavior obfuscation. We propose and implement such a framework that obfuscates the observable behavior of malware binaries. To assess the framework's utility, we use it to obfuscate known malware binaries and then investigate the impact on detection effectiveness of different n-gram based detection approaches. We find that the obfuscation transformations employed by FEEBO significantly affect the precision of such detection approaches. Several n-gram-based approaches can hence be concluded not to be resilient against this simple kind of obfuscation.

1 Introduction

Malware continues to be a relevant cyber security threat. While in the early days of the Internet malware was often developed for the pure sake of curiosity, malware development today follows a clear-cut business model. The motivations to develop and utilize malware ranges from supporting cyber espionage over theft of confidential data, denial-of-service of commercial services, or even blackmailing, up to tampering with military or civilian infrastructures.

Industry and academia continuously devise countermeasures to cope with this threat in form of advanced malware detection approaches. However, malware developers are often several steps ahead the state of the art. Most commercial antivirus software in principle continues to be some form of signature-based analysis on the persistent representation of potential malware. Not surprisingly, almost all modern malware families employ some means to confuse and hamper signature-based approaches. Such countermeasures range from simple techniques (e.g. build-time encryption and runtime decryption up), to more sophisticated techniques (e.g. control-flow obfuscation or anti-debugging mutations) [6].

Given control-flow obfuscation of today's malware, one intuitively appropriate detection strategy is so-called behavioral detection. The idea is to look at the malware's runtime behavior rather than its static code. This behavior includes issued function or system calls, or in general, every runtime interaction with system resources. By construction, behavioral detection approaches are barely affected by control-flow obfuscation. However, although behavioral detection techniques compensate the effects of (build-time) control-flow obfuscation techniques to a large extent, they are often vulnerable to more advanced (run-time) behavior obfuscation techniques that "blur" the externally visible behavior of malware. Examples for such behavior obfuscation techniques include the injection of bogus system calls or the deliberate randomized re-ordering of call execution sequences.

While control-flow obfuscation of malware and respective countermeasures at the detection side have been well researched [6], the effects of *behavior* obfuscation on the effectiveness of detection approaches so far only received very little attention in the literature. Behavior obfuscation in itself has been discussed from a theoretical perspective [7], but we are not aware of any empirical investigations of the effects of behavior obfuscation of real-world malware.

To provide a foundation for such empirical evaluations, we propose a behavior obfuscation framework which we call FEEBO. Provided an arbitrary malware sample as input, it applies a diverse set of behavior obfuscation transformations to its externally visible behavior, which is defined by issued system calls. This makes it possible to "inject" behavior obfuscation mechanisms into malware samples in a structured and targeted way, regardless of whether or not the specific malware sample performs any behavior obfuscation itself. Considering that behavior obfuscation at the system call level is still rarely done by real-world malware, this approach allows us to get one step ahead of malware developers and reason about the impact of such obfuscation techniques on state of the art detection approaches before they are implemented and released into the wild.

Contributions: **a)** To our best knowledge, we are the first to propose an empirical malware behavior obfuscation framework that is able to behaviorally obfuscate standard malware binaries. **b)** With FEEBO we establish a basis for a wide range of reproducible behavioral obfuscation resilience experiments. **c)** Our evaluations show that for certain configurations, the precision of n-gram [15] based detection approaches are significantly affected by behavioral obfuscation.

Organization: We introduce the concept of behavior obfuscation and discuss two main representative n-gram-based behavioral detection techniques in §2. Then we describe the design and implementation of FEEBO in §3. We show the effectiveness of a prototypical implementation of our framework and discuss its limitations in §4. We discuss possible application areas of our approach and give an outlook on future work in §5.

2 Preliminaries

We start with some relevant concepts from the literature. In particular we recall related work on behavior obfuscation and detection based on n-grams.

2.1 Behavior Obfuscation

This paper is inspired by the work of Péchoux and Ta [11] on behavior obfuscation of malware. They divide the behavior, i.e. executed operations of a program (e.g. malware) into (i) internal computations and (ii) system calls. *Internal computations* operate only on the process memory of the corresponding program and they only affect and are affected by the information stored inside this process' memory. *System calls* represent interactions with the operating system (OS) kernel, i.e. there is a transfer of control from the corresponding program to the kernel and back. Therefore, system calls affect and are affected by the information stored anywhere in the OS memory.

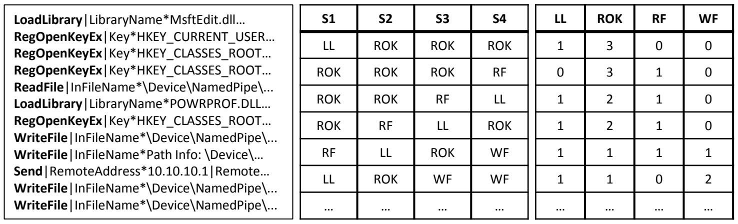
The sequence of system calls performed by a program is called the *observable (execution) path* or *behavior*. Péchoux and Ta show that it is possible to transform (obfuscate) the observable path of known malware samples such that the original malware functionality is preserved by: (i) inserting system calls before and/or after system calls in the observable path, (ii) reordering system calls in the observable path and (iii) substitution of system calls by other system calls which provide at least the same functionality. Different from our work, their goal is to obtain a trace that is similar to a goodware trace (mimicry). We, on the other hand, focus on randomly generating sets of malware "mutants" to assess their effect on behavioral detection approaches that analyze the system calls executed by malware.

There is an important difference between *behavior obfuscation* and *control-flow obfuscation*. Control-flow obfuscation applies transformations at the source code or intermediate representation levels in order to make a program harder to understand by a human or an automated analysis engine. Such code transformations include virtualization obfuscation, insertion of bogus code via opaque predicates, function splitting, and control-flow flattening [6]. These transformation will typically not have an effect on the observable execution path of that program. On the other hand, behavior obfuscation strictly implies changing the observable execution path of the program being obfuscated.

2.2 Behavioral Malware Detection

In contrast to approaches that focus on the persistent representation of malware, behavioral detection approaches discriminate malware from goodware by establishing characteristic behavior profiles. Such approaches range from using raw system call traces to short sequences of calls, so-called n-grams [9,14,15], to more elaborate concepts that model the semantic interdependencies between different calls in call-graphs [10,5,4]. There also exist approaches that model behavior by abstracting system calls into induced data flows [1,3]. These approaches are based on traces of issued system calls and are thus likely to be affected by the aforementioned behavior obfuscation transformations.

In this study we focus on approaches that base on n-grams as a behavior model, due to its prevalence in academic publications [15]. We are aware that findings based on this model do not necessarily generalize. Nevertheless we are convinced that such an evaluation is a good starting point to reason about the effects of behavior obfuscation in general and will be the basis for future work. To cover a broad range of n-gram based detection approaches, we follow the categorization schema of Canali et al. [2]. We consider n-grams built on system



```
LoadLibrary|LibraryName*MsftEdit.dll…
RegOpenKeyEx|Key*HKEY_CURRENT_USER…
RegOpenKeyEx|Key*HKEY_CLASSES_ROOT…
RegOpenKeyEx|Key*HKEY_CLASSES_ROOT…
ReadFile|InFileName*\Device\NamedPipe\…
LoadLibrary|LibraryName*POWRPROF.DLL…
RegOpenKeyEx|Key*HKEY_CLASSES_ROOT…
WriteFile|InFileName*\Device\NamedPipe\…
WriteFile|InFileName*Path Info: \Device\…
Send|RemoteAddress*10.10.10.1|Remote…
WriteFile|InFileName*\Device\NamedPipe\…
WriteFile|InFileName*\Device\NamedPipe\…
```

S1	S2	S3	S4
LL	ROK	ROK	ROK
ROK	ROK	ROK	RF
ROK	ROK	RF	LL
ROK	RF	LL	ROK
RF	LL	ROK	WF
LL	ROK	WF	WF
…	…	…	…

LL	ROK	RF	WF
1	3	0	0
0	3	1	0
1	2	1	0
1	2	1	0
1	1	1	1
1	1	0	2
…	…	…	…

Fig. 1: Call trace vs. ordered n-gram (a) vs. unordered n-gram (b)

calls without arguments as atoms and both a) considering or b) ignoring the ordering of calls for their construction. To test the aforementioned approaches we first executed known malware and goodware in a sandboxed environment and monitored their executed system calls. This procedure yielded labeled event logs, which we tokenized with a sliding window, moving a window of defined but fixed size over the respective log, thus yielding sets of n-grams of system calls.

For the first n-gram approach, which considers the ordering of system calls (a), we directly feed the obtained n-grams as features into a supervised machine learning classifier. For the second n-gram approach (b), which does not consider the ordering of system calls (b), we count the number of occurrences of each system call in the n-gram, build a feature vector with the number of occurrences of each of the system calls in the n-gram, and feed these vectors into the classifier. Note the independence of the feature vectors from the ordering of system calls in the n-gram.

Figure 1 depicts the resulting feature vectors for both approaches when applied to a small sample call trace (left). The middle shows n-grams for approach (a), consisting of 4 system calls on each row (i.e, 4-grams). The contents of the cells are the initials of the system calls from the trace to the left. The table on the right part shows n-grams for approach (b), which consist of the frequency of every system call (depicted in the table header) for a 4-gram on each row.

3 Our Approach to Behavior Obfuscation

Transforming (obfuscating) x86 binary programs without debugging symbols is a non-trivial task which involves binary rewriting [12]. This task becomes even more challenging when the binaries we want to transform are malware, which employ anti-disassembly techniques [8]. However, since we only want the binary to have a different observable behavior in terms of systems calls, we have taken an alternative approach by using binary instrumentation [13].

In a nutshell binary instrumentation allows one to intercept any system calls performed by the target binary. One can choose to execute, delay, drop or even swap the intercepted system call, plus perform other additional instructions including making more system calls. We have implemented the following two behavior obfuscation transformations: (i) system call *insertion* and (ii) system call *reordering*. These are relatively simple techniques in comparison to substitution of system calls with functionally equivalent systems calls; we leave their implementation to future work.

3.1 System Call Insertion

With a given probability p_i, system call insertion adds for each system call made by the obfuscated application a number of additional system calls randomly chosen from the previously executed system calls. The number of inserted system calls is randomly chosen between min_i and max_i, two more input parameters of FEEBO. To prevent these inserted calls from changing the original functionality of the application, we modify the values of their parameters in case the system calls belong to a set S of calls that have side-effects such as writing to a file. The values of the changed parameters are chosen such that they will not collide with existing data, e.g., files. Furthermore, system calls that access a unique system resource are excluded. For instance, if we were to insert the system call that sets the clipboard data, we would need to also insert a second call to restore the clipboard data since there is only one clipboard on each system.

For example, with $p_i = 0.25$, $min_i = 2$ and $max_i = 5$, every system call made by the application has a 25% chance to insert a randomly chosen number between 2 and 5 of system calls after the execution of the intercepted system call. This obfuscation transformation changes the externally visible behavior by inserting a random number of system calls in random locations of the original execution trace. The intuition is that it should be effective against n-grams-based detection approaches since they rely on patterns.

3.2 System Call Reordering

System call reordering can naïvely be implemented by delaying a sequence of system calls in a buffer which is randomly permuted before execution. This would most likely break the functionality of the transformed program or even cause it to crash. Instead, every system call in S executed by the transformed application, can be delayed with probability p_r and placed in a queue (of size n) for later execution. The reason only calls in S are being delayed is that calls outside S generally read information which applications need to continue their proper execution. Moreover, we use a queue for the delayed system calls, because we want to preserve the original ordering of system calls that have side effects like writing to a file. Once the queue reaches a certain size, our tool will execute them in their original order. Each of the delayed calls can additionally trigger the insertion other system calls with similar parameters as described in §3.1, i.e. probability of insertion denoted p_{ri} and the minimum and maximum number of inserted system calls, denoted by min_{ri}, respectively max_{ri}.

For example, for $p_r = 0.5$, $n = 5$, $p_{ri} = 0.75$, $min_{ri} = 1$ and $max_{ri} = 2$, every system call from S made by the application is delayed with a 50% probability. Once 5 calls have been delayed, they will be executed. Each of the delayed executions has a 75% probability to insert one or two other system calls.

3.3 Obfuscation Profiles

The range of the input parameters of the previously described obfuscation transformations are shown in Table 1. The insertion and reordering probabilities range

	$p_{\{i,ri\}}$	$min_{\{i,ri\}}$	$max_{\{i,ri\}}$	p_r	n
System Call Insertion	$[0,1]$	$\{0,\ldots,max_{\{i,ri\}}\}$	$\{min_{\{i,ri\}},\ldots,\infty\}$	–	–
System Call Reordering	$[0,1]$	$\{0,\ldots,max_{\{i,ri\}}\}$	$\{min_{\{i,ri\}},\ldots,\infty\}$	$[0,1]$	$\{0,\ldots,\infty\}$

Table 1: Obfuscation transformations versus parameters

from 0 to 1. The minimum and maximum numbers of inserted system calls as well as the size of the reordering queue are positive integers. Their upper bound depends on the data type and the architecture of the system they are running on. Based on these parameters of system call insertion and system call reordering we can configure various obfuscation profiles, e.g. "always insert 2 system calls after each system call in the original observable path", "do not insert any calls, only reorder" or "insert 1 system call after each reordered call". We will see concrete detection values for different obfuscation profiles in §4.

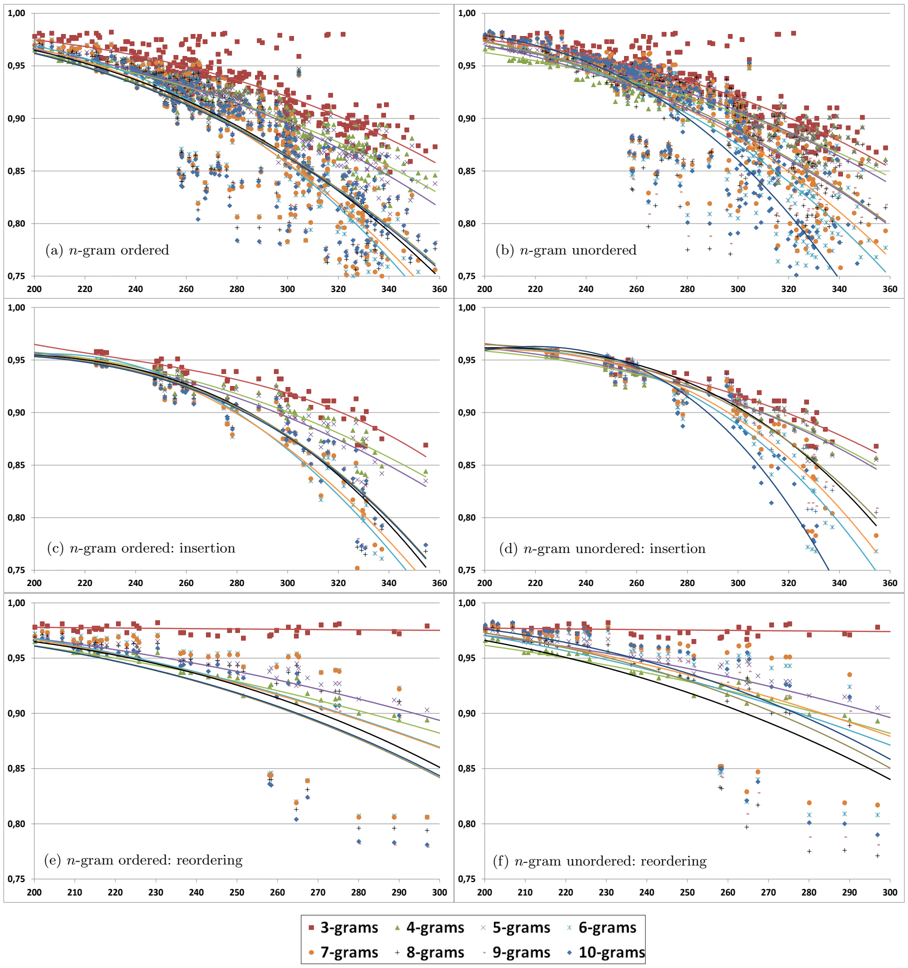
4 Evaluation

To assess the applicability of FEEBO, we obfuscated a set of real-world malware with the help of FEEBO and then applied the previously introduced behavior detection approaches, based on n-grams of system calls, to the resulting obfuscated system call traces.

Setup. We executed 100 malware samples within an installation of the Cuckoo malware analysis sandbox[1], where we replaced the behavior monitor with FEEBO to obtain a variety of obfuscated behavior traces of those samples. In addition, we collected the traces of 100 known goodware samples which we did not obfuscate, to use as comparison baseline for later training the detection classifiers.

The large range of values that the obfuscation parameters can take (see Table 1, quickly leads to a combinatorial explosion of the obfuscation profiles. Moreover, to capture a critical mass of system calls sufficiently large to allow training a classifier with good accuracy, we need to monitor a malware sample for at least 3 minutes. With one configuration profile capturing the obfuscated traces of 100 malware samples would then take 300 minutes which, with help of parallel execution of multiple VMs on 5 cores, we could cut down to about one hour per run. Therefore, we conducted experiments with 375 different combinations of the obfuscation parameters. More specifically, we set all probabilistic parameters like the insertion or reordering probability to selected values between 0% and 100%, i.e. $p_{\{i,r,ri\}} \in \{0.0, 0.25, 0.5, 0.75, 1.0\}$, and particular interesting discrete parameters to fixed values between 1 and 10, i.e. $max_i \in \{1, 5, 10\}$. All other parameters were set to fixed values, i.e. $min_{\{i,ri\}} = 1$, $max_{ri} = 3$ and $n = 5$. Conducting one evaluation for each configuration profile (e.g. one combination of the aforementioned parameters and value ranges) ends up in $5 \times 5 \times 5 \times 3 = 375$ runs, which sums up to a total runtime of about 16 days.

[1] http://www.cuckoosandbox.org/.



(a) n-gram ordered

(b) n-gram unordered

(c) n-gram ordered: insertion

(d) n-gram unordered: insertion

(e) n-gram ordered: reordering

(f) n-gram unordered: reordering

Fig. 2: Degree of Obfuscation (x-axis) versus Detection Rate (y-axis)

Experiments. Using the resulting execution traces, we trained the respective classifiers on the feature vectors computed on the non-obfuscated baseline traces and used the generated classifier on the remaining obfuscated event traces. For the ideal case of the applied obfuscations not having any effect on the externally visible behavior, the detection rate should remain 100%. With this setting we could investigate the effects of the applied obfuscation transformations with respect to detection accuracy. To assess the effects of different n-gram sizes we repeated this procedure for all possible n-grams for n between 3 and 10.

Figure 2 summarizes the experimental findings. As a measure of the degree of obfuscation, we calculated the *Levenshtein distance* between the respective traces, as it represents the number of atomic insertion, deletion, and substitution operations that are needed to transform one event trace into another one. For computing the Levenshtein distance we abstracted our traces to only the name of the system calls (not their parameters), which are elements of our alphabet. Correspondingly, the x-axis of each diagram represents the average obfuscation degree of all considered event traces, whereas the y-axis represents the detection rate (percentage of correctly identified malware samples) achieved by different detection approaches. To visualize the development of the median detection rate for increasing obfuscation degree we also plot trend-lines for each n-gram.

We split the evaluation results into three parts: the first row represents the results for the experiments where both type of obfuscation transformations, i.e. call reordering and call insertion were applied; the second row illustrates the results for the insertion experiments; and the last row the results from the reordering experiments. As we can see, the applied obfuscation transformations have a significant effect on the detection effectiveness of the n-gram approaches. In the first row of Figure 2 we can deduce a roughly quadratic relationship between an increase of obfuscation degree and an decrease of the detection rate. Also we can see, that the spread in classification accuracy, i.e. the standard deviation of the detection rate, significantly rises the more obfuscation is applied. Furthermore we can see that higher-order n-grams are more sensitive towards obfuscation.

Looking at the remaining diagrams we notice that insertion transformations seem to have a bigger impact on detection accuracy than reordering transformation, which is reflected in a significantly steeper slope of the trend-lines in the insertion diagrams than in the reordering diagrams. Also we can say that for very small n-gram sizes, reordering transformations seem to have barely any influence on the detection rate, as can be seen by almost constantly high detection rates. Finally, our evaluations did not reveal any significant difference in obfuscation resilience between the ordered and unordered types of n-gram approaches.

Discussion and threats to validity. First note that although we only conducted one execution run for one constellation of configuration parameter, the fact that several parameter configurations lead to a sample with a similar Levenshtein distance allows us to achieve a good saturation of the obfuscation spectrum. Given that we obtain 375 distinct sets of 100 obfuscated event traces, for each profile in our experiment, this gives a rather high density of 41 data-points in a range of 20 units on the x-axis in the first row from Figure 2, which correspond to the "both insertion and reordering" obfuscation profile. However, the density is 7 data-points in a range of 20 units for the second and third rows which correspond to "insertion-only", respectively "reordering-only" obfuscation profiles.

We intentionally did not mention false positive rates of the n-gram approaches in our evaluation, because they are not relevant for our experiments, since we do not change or obfuscate the set of goodware during our experiments. Currently our experimental setting assumes the presence of a certain ground truth, i.e. the availability of a critical mass of unobfuscated malware for classifier training. If malware developers start to make more use of behavioral obfuscation mechanism the availability of such a basic training set is not guaranteed. Using obfuscated malware for both, testing and training the classifiers, will likely diminish their effectiveness even more. For future work, we therefore also plan on investigating whether these factors impact the results.

Having performed some initial experiments with Naïve Bayes, Gaussian-kernel SVMs, and Random Forest classifiers, we can confirm that the choice of the baseline classifier does not have a significant effect on the relative obfuscation sensitivity of the considered n-gram approaches.

The functionality of any obfuscated program should include the functionality of the original (non-obfuscated) program. For many software transformation engines such as optimizing compilers, this is a strict requirement. However, even very widely used compilers such as GCC or Clang have been found to contain optimizations that break the functionality of the original source code [16]. The *system call reordering* transformation described above suffers from the same issue, i.e. it may change the functionality of malware such that it becomes ineffective. Arguably, however, in the case of obfuscating widely used malware it is more important to avoid detection even if the obfuscation engine will output some samples which are not effective. We do not yet possess statistics regarding the number of effective malware samples output by our tool. However we plan to study this fact as part of future work. We still consider our results valuable given that checking for behavioral equality in general is not decidable and in our experiments none of the obfuscated malware samples crashed during execution.

In sum, we can draw two main conclusions from our experiments: a) FEEBO is able to effectively obfuscate the behavior of real-world malware with significant effect on the effectiveness of behavioral detection approaches; b) the considered type of n-gram approaches is highly sensitive to the evaluated forms of behavior obfuscation.

5 Conclusions and Future Work

We have introduced FEEBO, a framework to conduct empirical experiments on the effects of behavior obfuscation on malware detections. To this extent we developed a prototype that can apply certain obfuscation transformations to the externally visible behavior of malware samples. To evaluate the effectiveness of the implemented obfuscation transformations and of our approach in general, we investigated the effects of a wide range of behavior obfuscation transformations on the detection capabilities of two representative n-gram behavior detection approaches. We could show that both types of n-gram approaches are considerably vulnerable to the applied obfuscation transformations.

We are aware that our presented evaluation results are not comprehensive in its present form. In particular, for future work we plan to repeat the experiments for a bigger configuration space and malware sets. We also plan to investigate the effects of lack of ground truth by training the classifiers on obfuscated malware samples instead on solely unobfuscated ones. In terms of possible extensions of FEEBO, we plan to implement additional obfuscation transformations that e.g. also tackle the substitution of certain system calls with semantically equivalent ones.

Although we release FEEBO[2] to parties from academia and industry, for ethical reasons we will provide a version that is not capable of generating self-contained obfuscated malware binaries. Instead, FEEBO needs to be manually installed in the evaluation environment, together with a installation of *Intel Pin* [13], which hopefully hampers misuse of FEEBO by malware developers.

References

1. S. Bhatkar, A. Chaturvedi, and R. Sekar. Dataflow anomaly detection. In *S&P*, pages 15–pp. IEEE, 2006.
2. D. Canali, A. Lanzi, D. Balzarotti, C. Kruegel, M. Christodorescu, and E. Kirda. A quantitative study of accuracy in system call-based malware detection. In *ISSTA '12*, pages 122–132. ACM, 2012.
3. L. Cavallaro and R. Sekar. Taint-enhanced anomaly detection. *Information Systems Security*, pages 160–174, 2011.
4. M. Christodorescu, S. Jha, and C. Kruegel. Mining specifications of malicious behavior. In *India Software Engineering Conference*, pages 5–14, 2008.
5. M. Christodorescu, S. Jha, S. Seshia, D. Song, and R. Bryant. Semantics-Aware Malware Detection. *S&P'05*, pages 32–46, 2005.
6. C. Collberg and J. Nagra. *Surreptitious Software: Obfuscation, Watermarking, and Tamperproofing for Software Protection*. Addison-Wesley Professional, 1st edition, 2009.
7. M. Dalla Preda, M. Christodorescu, S. Jha, and S. Debray. A semantics-based approach to malware detection. *ACM Transactions on Programming Languages and Systems*, 30(5):1–54, 2008.
8. C. Eagle. *The IDA pro book: the unofficial guide to the world's most popular disassembler*. No Starch Press, 2011.
9. S. Forrest, S. A. Hofmeyr, A. Somayaji, and T. A. Longstaff. A sense of self for unix processes. In *S&P '96*, pages 120–128, Washington, DC, USA, 1996. IEEE Computer Society.
10. Y. Park, D. S. Reeves, and M. Stamp. Deriving common malware behavior through graph clustering. *Computers & Security*, 2013.
11. R. Péchoux and T. D. Ta. A categorical treatment of malicious behavioral obfuscation. In *Theory and Applications of Models of Computation*, pages 280–299. Springer, 2014.
12. M. Prasad and T.-c. Chiueh. A binary rewriting defense against stack based buffer overflow attacks. In *USENIX Annual Technical Conference, General Track*, pages 211–224, 2003.
13. V. J. Reddi, A. Settle, D. A. Connors, and R. S. Cohn. Pin: a binary instrumentation tool for computer architecture research and education. In *Proceedings of the 2004 workshop on Computer architecture education*, page 22. ACM, 2004.
14. K. Rieck, P. Trinius, C. Willems, and T. Holz. Automatic analysis of malware behavior using machine learning. *J. of Computer Security*, pages 639–668, 2011.
15. C. Wressnegger, G. Schwenk, D. Arp, and K. Rieck. A close look on n-grams in intrusion detection: anomaly detection vs. classification. In *Workshop on Artificial intelligence and security*, pages 67–76, 2013.
16. X. Yang, Y. Chen, E. Eide, and J. Regehr. Finding and Understanding Bugs in C Compilers. *SIGPLAN Not.*, 46(6):283–294, June 2011.

[2] https://www22.in.tum.de/tools/feebo/